IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

Busca en Internet información sobre estas cuestiones:

- **Busca un par de ejemplos en Phyton de problemas como el de los misioneros y caníbales, o como el problema de lechero cuya solución se base en el paso entre distintos estados.**

Torres de Hanoi:

```
# coding=utf-8

import time


# Dibuja las torres.
def dibujarTorres():
    for fila in torres:
        for col in fila:
            if col == 0:
                print("            |            ", end="")
            elif col == 1:
                print("          [///]          ", end="")
            elif col == 2:
                print("        [/////]          ", end="")
            elif col == 3:
                print("      [///////]          ", end="")
            elif col == 4:
                print("    [/////////]          ", end="")
            elif col == 5:
                print("  [///////////]          ", end="")
            elif col == 6:
                print("[/////////////]          ", end="")
            elif col == 7:
                print("[///////////////]        ", end="")
        print()
    print("="*77)
    print("          1                         2                         3          ")
    time.sleep(1)

# Nos devuelve el disco de arriba de la columna col, sino devuelve 0.
def buscarDiscoArriba(col):
        fila = 0
        while fila <= discos and torres[fila][col] == 0:
                fila += 1
        if fila <= discos:
                return torres[fila][col]
        else:
                return 0

# Nos devuelve el espacio vacio de arriba de la columna col.
def buscarEspacioArriba(col):
        fila = 0
        while fila <= discos and torres[fila][col] == 0:
                fila += 1
        return fila - 1

# Elimina el disco de arriba de la columna col.
def eliminarDiscoArriba(col):
    fila = 0
    while fila <= discos and torres[fila][col] == 0:
        fila += 1
    torres[fila][col] = 0

# Representación gráfica.
def hanoiGrafico(n, origen=1, auxiliar=2, destino=3):

    if n > 0:
        hanoiGrafico(n-1, origen, destino, auxiliar) # n-1 discos de la torre origen a la torre auxiliar.
        disco = buscarDiscoArriba(origen-1)
        eliminarDiscoArriba(origen-1)
        torres[buscarEspacioArriba(destino-1)][destino-1] = disco
        print("\n"*40)
        dibujarTorres()
        hanoiGrafico(n-1, auxiliar, origen, destino) # n-1 discos de la torre auxiliar a la torre final.

# Representación en modo texto.
def hanoiTexto(n, origen=1, auxiliar=2, destino=3):

    if n > 0:
        hanoiTexto(n-1, origen, destino, auxiliar) # n-1 discos de la torre origen a la torre auxiliar.
        print("Se mueve el disco %d de torre %d a la torre %d" % (n, origen, destino)) # disco n a la torre destino.
        hanoiTexto(n-1, auxiliar, origen, destino) # n-1 discos de la torre auxiliar a la torre final.

print("\n"*40)
print("     TORRES DE HANOI     ")
print("*"*25)
modo = int(input("Ingrese la opcion deseada:\n\n1) Modo grafico\n2) Modo texto\n\n"))

if modo == 1:
    discos = int(input("\nIngrese entre 1 y 7 discos: "))

    # Defino la matriz para el gráfico
    if discos > 0 and discos < 8:
        if discos == 1:
            torres = [[0,0,0],[1,0,0]]
        elif discos == 2:
            torres = [[0,0,0],[1,0,0],[2,0,0]]
        elif discos == 3:
            torres = [[0,0,0],[1,0,0],[2,0,0],[3,0,0]]
        elif discos == 4:
            torres = [[0,0,0],[1,0,0],[2,0,0],[3,0,0],[4,0,0]]
        elif discos == 5:
            torres = [[0,0,0],[1,0,0],[2,0,0],[3,0,0],[4,0,0],[5,0,0]]
        elif discos == 6:
            torres = [[0,0,0],[1,0,0],[2,0,0],[3,0,0],[4,0,0],[5,0,0],[6,0,0]]
        elif discos == 7:
            torres = [[0,0,0],[1,0,0],[2,0,0],[3,0,0],[4,0,0],[5,0,0],[6,0,0],[7,0,0]]

        print("\n"*40)
        dibujarTorres()
        hanoiGrafico(discos)
    else:
        print("\nERROR! Solo se permiten de 1 a 7 discos para el modo grafico.")
elif modo == 2:
    discos = int(input("\nIngrese numero de discos: "))

    if discos > 0:
        print()
        hanoiTexto(discos)
    else:
        print("\nERROR! Ingrese un numero mayor a 0.")
else:
    print("\nERROR! La opcion ingresada es incorrecta.")
```

El lobo, la cabra y la col

```
class WGC_Node:
    incompatibilities = [
        ["c", "g", "w"],
        ["g", "w"],
        ["c", "g"]
    ]

    def __init__(self, west=["w", "c", "g"], east=[], boat_side=False, children=[]):
        self.west = west
        self.east = east
        self.boat_side = boat_side
        self.children = children

    def __str__(self):
        return str(self.west) + str(self.east) + ("Left" if not self.boat_side else "Right")

    def generate_children(self, previous_states, parent_map):
        children = []
        if not self.boat_side:
            for i in self.west:
                new_west = self.west[:]
                new_west.remove(i)
                new_east = self.east[:]
                new_east.append(i)
                if sorted(new_west) not in WGC_Node.incompatibilities and not WGC_Node.state_in_previous(previous_states, new_west, new_east, not self.boat_side):
                    child = WGC_Node(new_west, new_east, not self.boat_side, [])
                    children.append(child)
                    parent_map[child] = self
            if sorted(self.west) not in WGC_Node.incompatibilities and not WGC_Node.state_in_previous(previous_states, self.west[:], self.east[:], not self.boat_side):
                child = WGC_Node(self.west[:], self.east[:], not self.boat_side, [])
                children.append(child)
                parent_map[child] = self
        else:
            for i in self.east:
                new_west = self.west[:]
                new_west.append(i)
                new_east = self.east[:]
                new_east.remove(i)
                if sorted(new_east) not in WGC_Node.incompatibilities and not WGC_Node.state_in_previous(previous_states, new_west, new_east, not self.boat_side):
                    child = WGC_Node(new_west, new_east, not self.boat_side, [])
                    children.append(child)
                    parent_map[child] = self
            if sorted(self.east) not in WGC_Node.incompatibilities and not WGC_Node.state_in_previous(previous_states, self.west[:], self.east[:], not self.boat_side):
                child = WGC_Node(self.west[:], self.east[:], not self.boat_side, [])
                children.append(child)
                parent_map[child] = self
        self.children = children

    @staticmethod
    def state_in_previous(previous_states, west, east, boat_side):
        return any(
            sorted(west) == sorted(i.west) and
            sorted(east) == sorted(i.east) and
            boat_side == i.boat_side
            for i in previous_states
        )


def find_solution(root_node, use_bfs=False):
    '''
    Find a solution to the WGC Problem.
    use_bfs: False for DFS, True for BFS
    '''
    to_visit = [root_node]
    node = root_node
    previous_states = []
    parent_map = {root_node: None}
    while to_visit:
        node = to_visit.pop()
        if not WGC_Node.state_in_previous(previous_states, node.west, node.east, node.boat_side):
            previous_states.append(node)
        node.generate_children(previous_states, parent_map)
        if use_bfs:
            to_visit = node.children + to_visit
        else:
            to_visit = to_visit + node.children
        if sorted(node.east) == ["c", "g", "w"]:
            solution = []
            while node is not None:
                solution = [node] + solution
                node = parent_map[node]
            return solution
    return None

if __name__ == "__main__":
    root = WGC_Node()
    solution = find_solution(root, use_bfs=False)
    print("DFS solution = [", end='')
    for i in solution:
        print(i, '\b, ', end='')
    print("\b\b]")

    solution = find_solution(root, use_bfs=True)
    print("BFS solution = [", end='')
    for i in solution:
        print(i, '\b, ', end='')
    print("\b\b]")
```

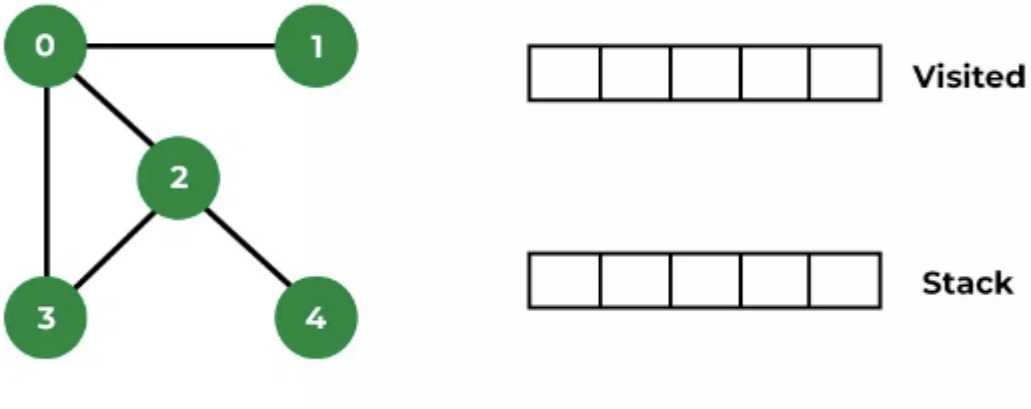
- **¿Por qué un grafo no resulta óptimo en los problemas de búsqueda?**

  Que un grafo séa mas o menos eficiente en los problemas de búsqueda depende mucho del algoritmo de busqueda que se utilice y de como ha sido el problema modelado en el grafo, es decir que hay varios aspectos que pueden afectar a la eficiencia en los problemas de busqueda, por ejemplo la forma en la que se representan los nodos y las aristas en el grafo puede afectar al rendimiento de los algoritmos de busqueda, la propia eleccion del algoritmo de busqueda es muy relevante en cuanto a tener una mejor eficiencia. En resumen un grafo en si mismo no tiene porque no resultar optimo en los problemas de búsqueda, ya que la eficiencia depende de lo bien representado que este el problema en el grafo, de la elección del algoritmo de búsqueda y de como sea la complejidad del problema específico.

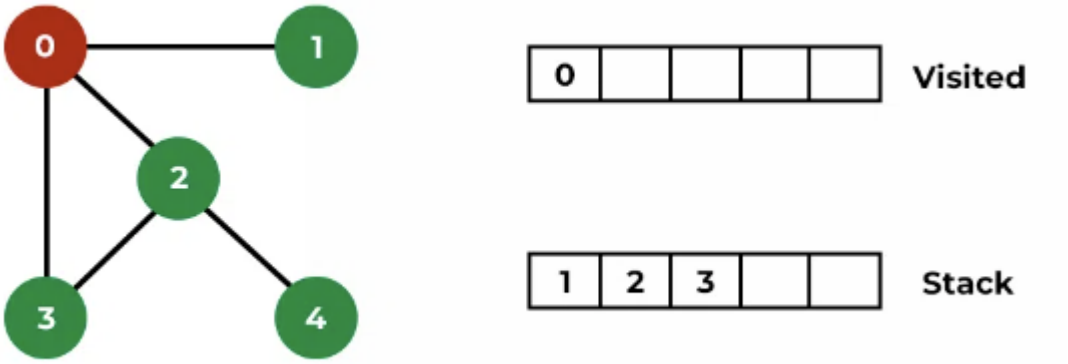
- **En las estrategias de búsqueda a ciegas, elige una, busca su algoritmo y una breve implementación en java, Python y en otro lenguaje que quieras.**

  Algoritmo de búsqueda en profundidad (DFS Depth first search).

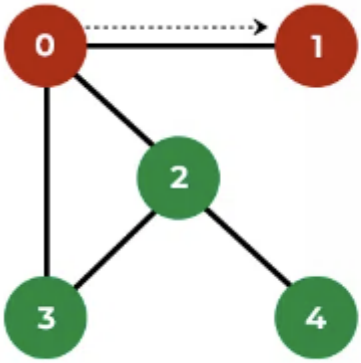
  Paso 1: El stack inicial y el array de nodos visitados estan vacíos.

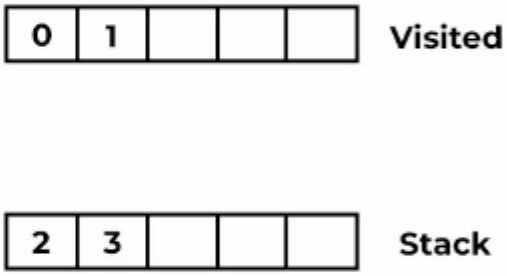


  Paso 2: Visita al nodo 0 y se ponen en el stack los nodos adyacentes

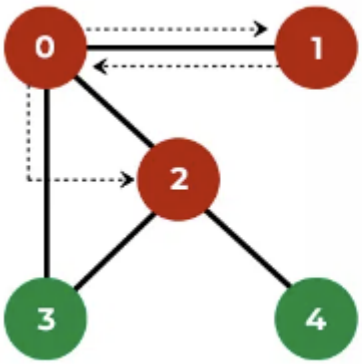


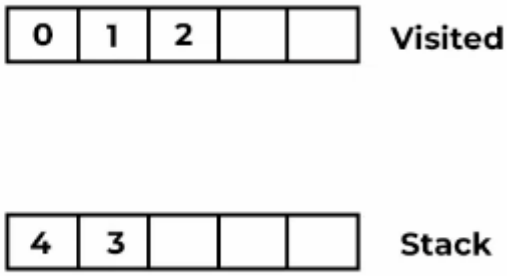
  Paso 3: Ahora se visita el nodo de la parte superior del stack y se añaden al stack los nodos adyacentes no visitados, en este caso ninguno.



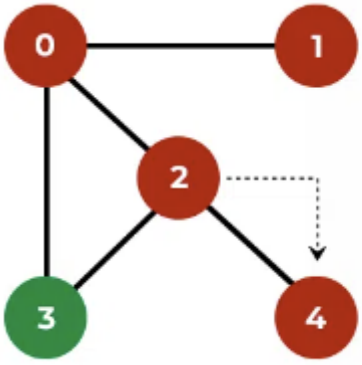


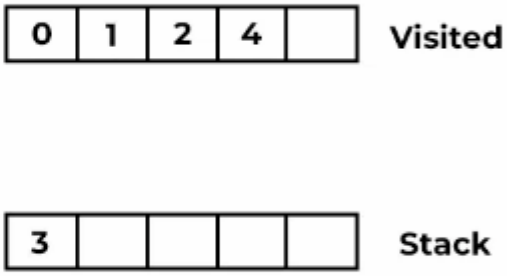
Paso 4: Ahora se visita el nodo 2 de la parte superior del stacky se ponen los nodos adyacentes no visitados en el stack, en este caso 4 y 3.



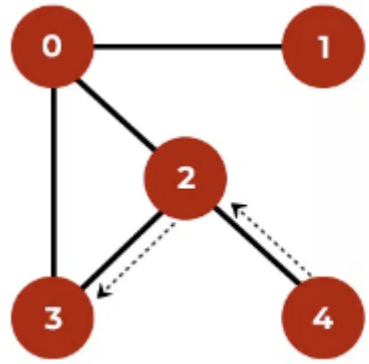


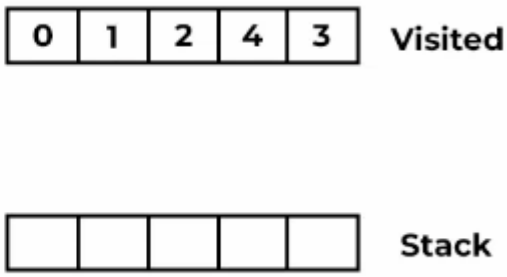
Paso 5: Ahora se visita el nodo 4 de la parte superior del stack y se ponen los nodos adyacentes no visitados en el stack, ninguno en este caso.





Paso 6: Ahora se visita el nodo 3, se añaden los nodos adyacentes no visitados al stack, en este caso ninguno y finaliza la búsqueda en profundidad.





Implementacion en Java

```java
import java.io.*;
import java.util.*;
class Graph {
    private int V;

    private LinkedList<Integer> adj[];

    @SuppressWarnings("unchecked") Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }

    void addEdge(int v, int w)
    {
        adj[v].add(w);
    }
    void DFSUtil(int v, boolean visited[])
    {
        visited[v] = true;
        System.out.print(v + " ");

        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }
    void DFS(int v)
    {
            boolean visited[] = new boolean[V];
            DFSUtil(v, visited);
    }

        public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println(

        g.DFS(2);
    }
}
```

Implementacion en Python

```python
from collections import defaultdict

class Graph:
    def __init__(self):

      self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):

        visited.add(v)
        print(v, end=' ')

        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    def DFS(self, v):

        visited = set()

        self.DFSUtil(v, visited)

if __name__ == "__main__":
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    print("Following is Depth First Traversal (starting from vertex 2)")

    g.DFS(2)
```

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

Implementacion en Javascript

```
class Graph
{
    constructor(v)
    {
        this.V = v;
        this.adj = new Array(v);
        for(let i = 0; i < v; i++)
            this.adj[i] = [];
    }

    addEdge(v, w)
    {
        this.adj[v].push(w);
    }

    DFSUtil(v, visited)
    {
        visited[v] = true;
        console.log(v + " ");
        for(let i of this.adj[v].values())
        {
            let n = i
            if (!visited[n])
                this.DFSUtil(n, visited);
        }
    }
    DFS(v)
    {
        let visited = new Array(this.V);
        for(let i = 0; i < this.V; i++)
            visited[i] = false;

        this.DFSUtil(v, visited);
    }
}
g = new Graph(4);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);
console.log("Following is Depth First Traversal " +
            "(starting from vertex 2)");
g.DFS(2);
```

- **En las estrategias de búsqueda con estrategia heurística, elige una, busca su algoritmo y una breve implementación en java, C y en otro lenguaje que quieras.**

  Busqueda A*, es un algoritmo de búsqueda informada utilizada comúnmente en problemas de búsquedas de caminos y optimización. A* combina la eficiencia de las búsquedas informadas (Heurística) con la garantía de encontrar la solución mas óptima. Su algoritmo podría ser descrito de la siguiente manera:

  1) Colocar el nodo inicial en una lista (ABIERTOS)
  2) Verificar si la lista de ABIERTOS está vacia.
     a. Si esta vacía entonces no se ha encontrado una solución. Fin
     b. Si no está vacia entonces hay que realizar los cálculos de h, g y f correspondientes para obtener el nodo con el mejor valor en f (el de menor costo) de la lista de ABIERTOS y se adiciona a otra lista (CERRADOS).

        f(x)= g(x) + h(x) donde g(x) es el costo real acumulado desde el inicio hasta el nodo x y h(x) es una función heurística que proporciona una estimación del costo restante desde el nodo x hasta el objetivo.
  3) Evaluar si el nodo que se acaba de enviar a CERRADOS es el objetivo.
     a. Si es el nodo objetivo entonces se encontró la solución.
     b. En caso de no ser el nodo objetivo se prosigue con las siguientes operaciones.

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

4) El nodo sucesor ahora se toma como el nodo actual.
5) Calcular los valores de g, h y f del nodo.
6) ¿El nodo actual posee mejor valor en f que su nodo padre?
    a. Si posee mejor valor entonces, retira de la lista de cerrados. Añade sus sucesores (nodos hijos) a la lista de abiertos y regresa al punto 2.
    b. Si no, termina la operación y regresa a evaluar la condición de si la lista de ABIERTOS se encuentra vacia.

- **Busca algoritmos asociados a los juegos como el minimax y descríbelos brevemente.**

Monte Carlo Tree Search (MCTS): Es un algoritmo de búsqueda basado en la simulación y el muestreo estadístico. Es especialmente efectivo en juegos con información imperfecta y decisiones secuenciales, como juegos de tablero. Las principales fases del MCTS son:

1) Selección: Comienza en el nodo raíz del árbol y se mueve hacia abajo seleccionando nodos de acuerdo con ciertas reglas.
2) Expansión: Cuando alcanza un nodo no explorado se expande creando uno mas nodos hijos que representan posibles acciones desde ese estado.
3) Simulación: Se realiza una simulación desde el nodo recién creado o un nodo existente seleccionando al azar. La simulación avanza hasta llegar a un estado terminal o alcanzar un limite de profundidad.
4) Retropropagación: La información del resultado de la simulación se retropropaga hacia arriba a lo largo de la ruta que llevo a la selección del nodo original. Se acutalizan las estadísticas de cada nodo en esa ruta.

Estas 4 fases se repiten en ciclos hasta que se alcanza un limite de tiempo o se realiza un número predeterminado de simulaciones. A medida que mas simulaciones son realizadas, el árbol de búsqueda se construye y refina y así las decisiones optimas son cada vez mas evidentes.

Q-Learning: Es un algoritmo de aprendizaje por refuerzo basado en la idea de aprendizaje mediante ensayo y error. Para ello busca la forma de maximizar el valor esperado de las recompensas, para ello el agente aprende a estimar el valor de cada acción posible en un estado especifico. Estos valores se almacenan en una tabla conocida como Q-table, que es como un mapa que asocia cada estado con todas las acciones posibles y sus posibles ganancias. Este algoritmo sigue un proceso de aprendizaje iterativo basado en la ecuación de Bellman, que expresa el valor optimo de una política como la suma de las recompensas inmediatas y el valor esperado de las recompensas futuras.

El proceso de aprendizaje se puede dividir en fases:

1. Inicializacion de la Q-table. Se establecen todos los valores a 0 o en un valor aleatorio. Esto representa el desconocimiento inicial del agente sobre la calidad de sus acciones.
2. Exploración de entorno: El agente comienza a explorar el entorno y a realizar acciones aleatorias, con la finalidad de recopilar datos del entorno.
3. Actualización de la Q-table: Después de realizar una acción en un estado específico, el agente recibe una recompensa y observa el nuevo estado en el que se encuentra. Actualiza el valor en la Q-table.
4. Selección de acciones optimas: Luego de haber explorado el entorno lo suficiente y actualizado la Q-table, el agente podrá comenzar a seleccionar acciones optimas. Estas se elegirán en función de los valores mas altos para maximizar así la recompensa.
5. Aprendizaje continuo: Cada nueva interacción con el entorno ayuda a refinar el conocimiento del agente y a mejorar su capacidad de tomar decisiones optimas.